Curso: Microsserviços na prática com Spring Cloud
Instrutor: Thiago Hernandes de Souza

Prefácio

Curso: Microsserviços na prática com Spring Cloud
Instrutor: Thiago Hernandes de Souza

1 – Introdução

Muita gente trabalha ou utiliza o Spring Boot e suas variantes para o desenvolvimento de sistemas. No entanto, muitos desconhecem a biblioteca **Cloud** do Spring.

Aparentemente soa como um termo correlacionado com "nuvens", no entanto, hipótese falso! Spring Cloud está diretamente ligado com o desenvolvimento em ambientes distribuídos. Objetivamente dizendo, proporciona o desenvolvimento de sistemas utilizando arquiteturas modernas como é o caso de **microsserviços**.

Os sistemas legados ou antigos mais conhecidos têm em quase toda sua plenitude arquiteturas monolíticas geralmente com orquestração mais complexa devido a sistemas fortemente acoplados e direcionados para um mesmo banco de dados, ou então, que use um barramento e canais únicos para alimentação e consulta de dados.

A arquitetura de microsserviços é uma evolução que tem como um de seus elementos a **descentralização** de códigos, rotinas, funções, módulos e APIs, possibilitando a quebra de uma aplicação "gigantesca", em diversos núcleos menores isolados, mas que conseguem se comunicar. Isso sem dúvida é uma tendência em que ocorre uma escalabilidade horizontal de recursos, o que facilita principalmente a manutenção e alterações em módulos distintos de uma aplicação.

2 – Entendendo o Spring Cloud

Arquiteturas que utilizam o Spring Cloud estão amparadas por três elementos primordiais: **escalabilidade**, **durabilidade** e **disponibilidade**. Esses, pressupõem características importantes para as aplicações modernas atuais o que consequentemente devem envolver microsserviços.

O core do Spring Cloud é composto basicamente conforme ilustração da imagem abaixo:

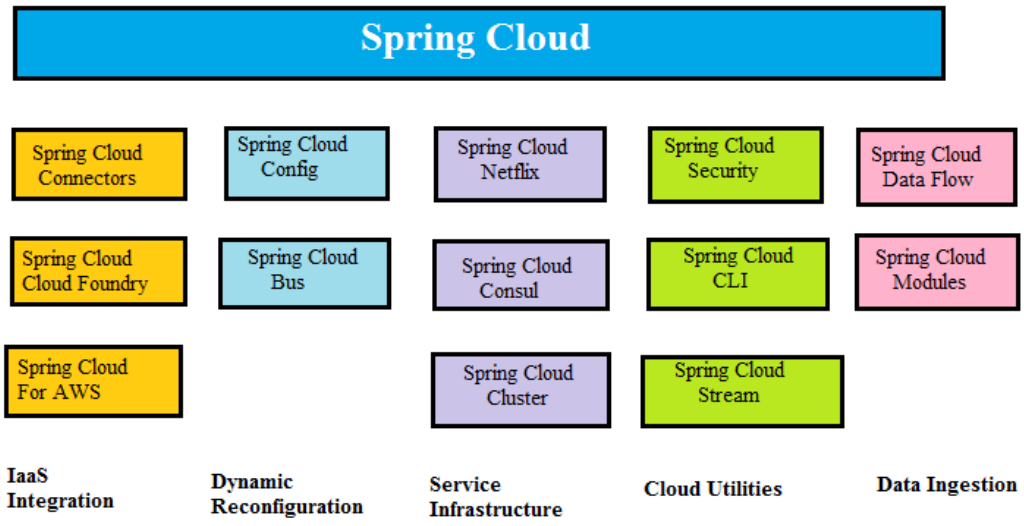

Figura 1 – Divisão do core do Spring Cloud

Cada qual com uma função específica, para quem quiser se aprofundar nos tópicos/teoria, além de exemplos e demais elementos, é possível consultar em: https://spring.io/projects/spring-cloud.

Baseado em seu uso, é importante salientar o uso e extensibilidade que o core do Spring Cloud possibilita:

- **Configuração distribuída;**
- **Serviços de registro e descoberta;**
- **Roteamento inteligente;**
- **Mensageria distribuída;**
- **Balanceamento;**
- **Serviços de disponibilidade e**
- **Inteligência em eleger novas instâncias.**

Sem dúvida teríamos inúmeros pontos teóricos e técnicos para tratar, mas prefiro avançar colocando a prática por meio da codificação e mostrando a **base necessária** para o desenvolvimento de aplicações utilizando o Spring Cloud.

2.1 – Spring Cloud Netflix

Netflix é um subprojeto do Spring Cloud. Nesse contexto ocorre a integração entre o **Netflix OSS** (Open Source Software) e o Spring Boot por meio de uma auto-configuração.

Os componentes do Netflix OSS estão voltados para o auxílio no desenvolvimento de sistemas distribuídos. São Eles:

- **Eureka – Service Discovery:** servidor com função específica de registro e comunicação entre de serviços**;**
- **Circuit Breaker – Hystrix:** consiste em um design pattern (padrão de projeto) de desenvolvimento que ajuda a evitar falhas em cascata**;**
- **Intelligent Routing – Zuul:** faz o gerenciamento de rotas entre as instâncias e
- **Client-Side Load Balacing – Ribbon:** realiza o balanceamento das instâncias registradas**.**

3 – Aprendendo com a Prática

Quanto a parte teórica suficientemente trouxe um facilitador para o leitor. Certamente, a partir desse capítulo, além da teoria ser trocada pela prática, deixará em evidência o quanto desenvolver projetos facilita a correlação entre teoria e prática.

3.1 – Primeiro Projeto

Em nosso primeiro projeto, criarei um "hello world" um pouco mais aperfeiçoado para o entendimento do Eureka Server e as ligações existentes entre os clientes, seja por meio de chamadas Feign ou Ribbon.

Ribbon: é um balanceador de carga (load balancer) que permite o controller sobre comportamentos ocorridos sobre os protocolos: HTTP e TCP.

Feign: é um web service client que utiliza em seu core o Ribbon como balanceador de carga.

No entanto, as chamadas do tipo Feign são um pouco mais simples do que o Ribbon. O que os diferem são as propriedades e parametrização.

3.1.1 – Protótipo da Arquitetura

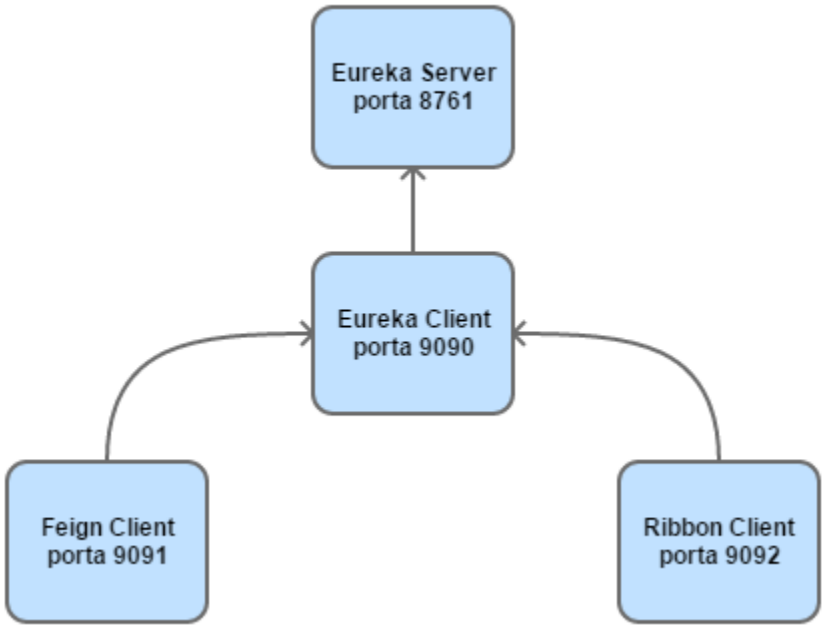

Figura 1 – Protótipo do primeiro projeto

Composto basicamente do servidor de registros de aplicações e três clientes, cada qual em sua respectiva porta. As chamadas do Feign Client e Ribbon Client farão menção de um controller no Eureka Client.

As chamadas são:

http://localhost:9090/api-client/ola – serviço disponibilizado para os clients Feign (9091) e Ribbon (9092)
http://localhost:9091/api-feign/ola – serviço Feign que acessa "GetMapping" do client (9090)
http://localhost:9092/api-ribbon/ola – serviço Ribbon que acessa "GetMapping" do client (9090)

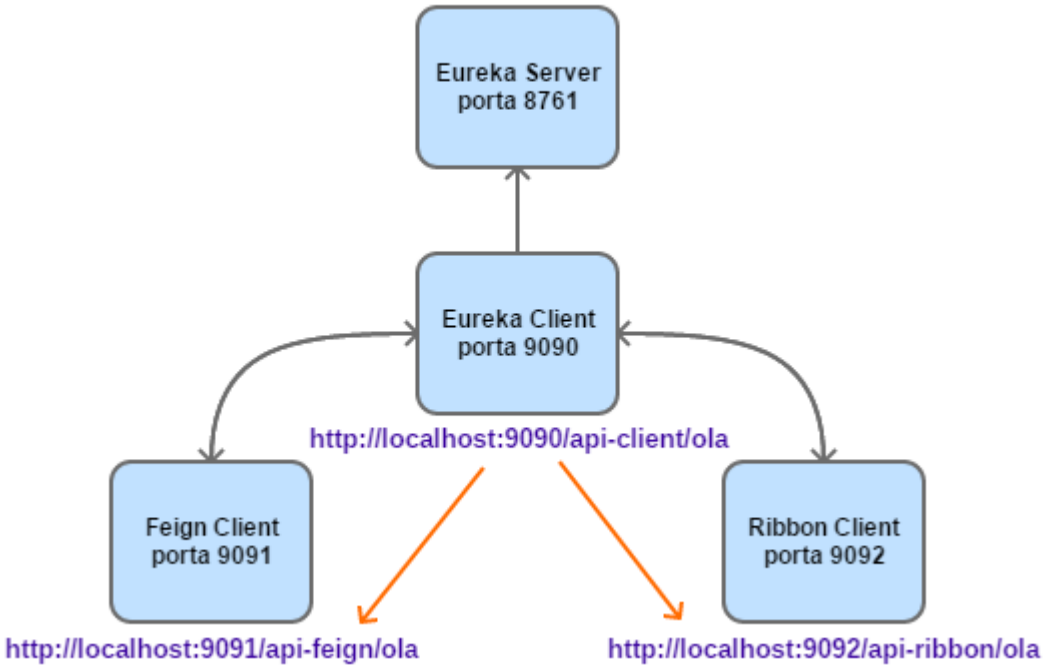

Figura 2 – Interação entre os serviços

3.2 – Segundo Projeto

Nesse segundo projeto o **"roteamento inteligente"** e a **segurança** serão explanados de modo a demonstrar os elementos de: disponibilidade e escalabilidade que estão introduzidos nessa arquitetura fabulosa de microsserviços.

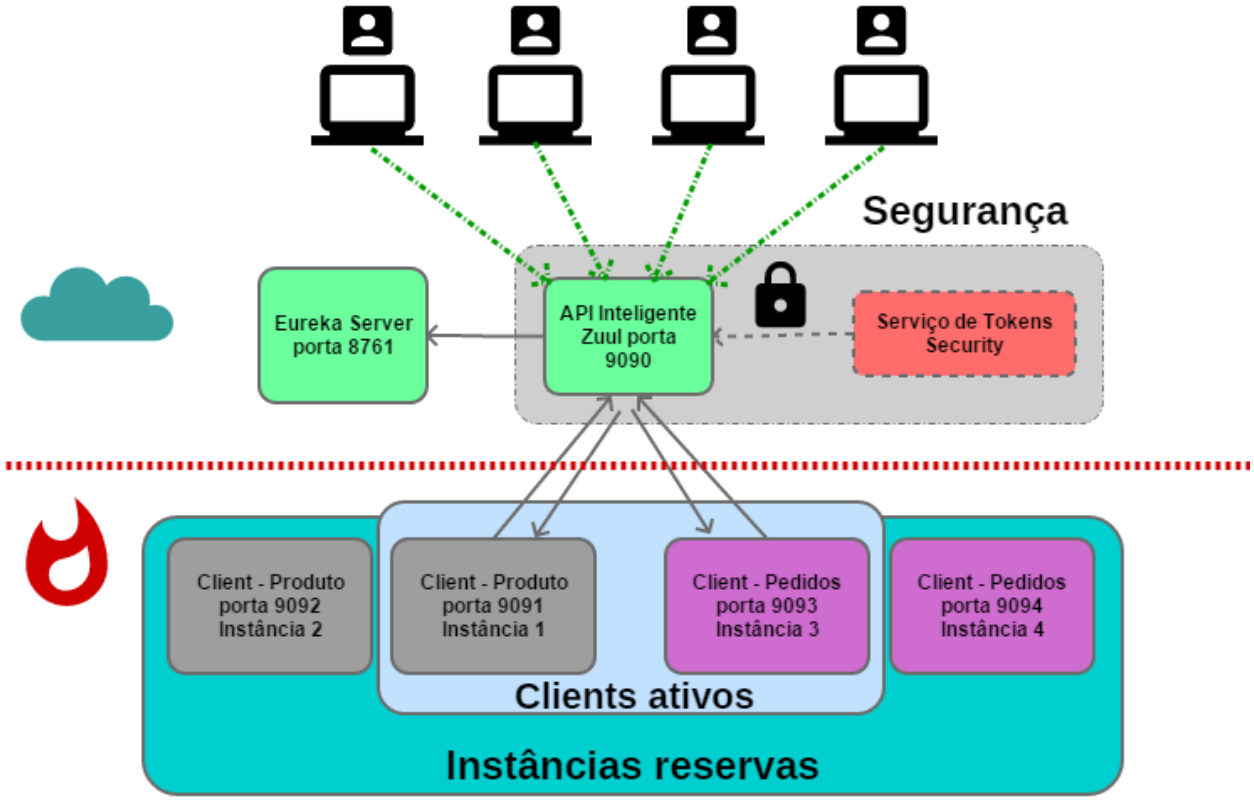

Figura 3 – Arquitetura do projeto de roteamento inteligente e segurança

Eureka Server porta 8761: servidor Eureka.

API Inteligente Zuul porta 9090: gateway de redirecionamento de rotas e **Serviço de Tokens** que são responsáveis pela criação de tokens de segurança.

Client – Produto porta 9091 – Instância 1: instância ativa de serviço de produtos.

Client – Produto porta 9092 – Instância 2: instância reserva/disponível para eleição caso ocorra falha da instância da porta 9091.

Client – Pedidos porta 9093 – Instância 3: instância ativa de serviço de pedidos.

Client – Produto porta 9094 – Instância 4: instância reserva/disponível para eleição caso ocorra falha da instância 9093.

3.2.1 – Serviço de Segurança

A segurança é extremamente importante em uma aplicação. Mas o que acontece nesse cenário de microsserviços? Na maioria dos casos implementar algum padrão de segurança ou até mesmo um personalizado, é muito mais prático para uma única aplicação.

Em nosso cenário seria viável implementar segurança em todos os microsserviços independentes? Poderia até ser feito, mas considerando elementos como de redução de códigos repetidos, reusabilidade e padronização no desenvolvimento, a grande "charada" nesse contexto é utilizar o microsserviço inteligente de API de roteamento.

É justamente nele em que ocorre o gerenciamento de rotas, eleição de novas instâncias e o filtro de requisições.

Sendo assim, a implementação consequentemente será na API Zuul Gateway.

Seguindo padrões adotados por inúmeras empresas, o **JWT** (JSON Web Token - https://jwt.io/) foi implementado em nossa arquitetura.

3.2.2 – API Zuul Gateway

Na API inteligente de roteamento o pacote que contém todos os arquivos referentes a validação e autenticação é o "**com.coder.deploy.security.jwt**". Quanto a questão de usuários, foi utilizado "inMemory", sendo que, para uma aplicação um pouco mais profissional, faria sentido determinar roles e retornar usuários de um banco de dados específico.

No entanto, para fins didáticos e para agilizar o processo, até porque o enfoque nesse ponto não é trabalhar com banco de dados, mas sim mostrar o funcionamento da arquitetura de microsserviços utilizando o Eureka e seus componentes.

3.2.3 – Disponibilidade de Instâncias

Ao observar a arquitetura na Figura 3 conseguimos identificar duas instâncias reservas: 9092 (produtos) e 9094 (pedidos). Elas são geradas a partir dos sistemas independentes de produtos e pedidos apenas com a alteração

da porta no arquivo localizado em "**resources**", "**application.yml**".

Essas instâncias podem ser uma ou mais, e possibilitam a escalabilidade e disponibilidade do funcionamento da arquitetura que foi definida.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-INTELIGENTE-ZUUL	n/a (1)	(1)	UP (1) - DESKTOP-UCD5TPH:API-INTELIGENTE-ZUUL:9090
PEDIDO-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-UCD5TPH:PEDIDO-SERVICE:9093
PRODUTO-SERVICE	n/a (2)	(2)	UP (2) - DESKTOP-UCD5TPH:PRODUTO-SERVICE:9092 , DESKTOP-UCD5TPH:PRODUTO-SERVICE:9091

Figura 4 – Exemplo real com instâncias em execução. Nesse caso apenas uma instância reserva de produtos disponível para assumir em caso da indisponibilidade da instância 9091 de produto.

3.2.4 – Circuit Breaker

Esse "pattern" de projeto que objetiva mitigar falhas de chamadas é extremamente importante para não interromper possíveis execuções assíncronas ou síncronas.

Para tal, foi feita uma implementação no **app de pedidos**. No caso de uma chamada de um produto que é oriundo do **app de produtos**, caso o mesmo falhe, podemos imaginar que poderia existir outros métodos em que seguiriam o término de um fluxo específico.

O detalhe mais interessante, sem dúvida, é o processo em que é restabelecido o serviço de produtos e não é emitido nenhum erro ou interrupção de chamadas ou execução do contexto do app web (conjunto de todos os microsserviços).